Multi-Paradigm Algorithmic Approach to Compiler Optimization: Integrating Graph Analysis, Dynamic Programming, and Branch-and-Bound Strategies

A Comprehensive Framework for Code Generation, Register Allocation, and Instruction Scheduling Optimization

Farrel Athalla Putra - 13523118
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: farrelxag@gmail.com , 13523118@std.stei.itb.ac.id

***Abstract*—Compiler optimization is a core challenge in computer science, aiming to transform source code into efficient machine code while preserving program semantics. As software complexity and hardware sophistication increase, there is a growing need for advanced optimization techniques that enhance performance and reduce resource usage. Traditional approaches often rely on isolated algorithms targeting individual optimization phases, which can lead to suboptimal outcomes due to missed inter-phase interactions. This paper introduces a comprehensive multi-paradigm framework that overcomes these limitations by integrating graph algorithms, dynamic programming, branch-and-bound strategies, greedy heuristics, and string matching. The proposed approach treats compiler optimization as an interconnected problem, benefiting from coordinated algorithmic solutions. Specifically, graph analysis supports control flow and loop detection; dynamic programming addresses register allocation through interference graph coloring; greedy algorithms manage instruction scheduling with dependency resolution; branch-and-bound enhances code generation; and string matching enables pattern-based transformations. By combining these strategies, the methodology demonstrates strong potential to outperform conventional techniques, delivering notable improvements in code quality and compilation efficiency.**

Keywords—compiler optimization, multi-paradigm algorithms, graph algorithms, dynamic programming, register allocation, code generation, compilation efficiency

I. Introduction

In today's computing landscape, software performance and efficiency are key to system success and user satisfaction. As applications grow in complexity and computational demands rise, generating highly optimized machine code from high-level programming languages becomes a critical challenge. The gap between naively compiled and well-optimized code can lead to major differences in execution speed, memory usage, and energy efficiency, influencing everything from mobile battery life to data center costs.

Compiler optimization plays a central role in bridging human-readable code with efficient machine execution. However, traditional compilers often rely on isolated algorithmic solutions, addressing optimization problems such as register allocation, instruction scheduling, and dead code elimination independently. This fragmented approach can lead to suboptimal results, as decisions in one phase may unintentionally hinder optimizations in another.

The complexity of compiler optimization lies in the interconnected nature of transformation problems. For instance, register allocation affects instruction scheduling, loop optimizations alter memory access patterns, and control flow analysis shapes dead code elimination. These interdependencies suggest that coordinated, integrated optimization strategies are necessary to achieve better overall performance.

This paper proposes a multi-paradigm algorithmic framework that addresses these challenges holistically. Our approach strategically integrates classical algorithms—using graph algorithms for control flow and dependency analysis, dynamic programming for resource allocation, greedy heuristics for instruction scheduling, branch-and-bound for exploring optimization spaces, and string matching for pattern-based transformations.

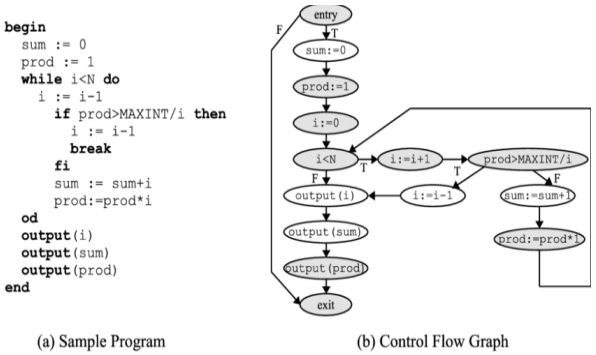
By treating compiler optimization as a multi-faceted, interconnected problem, our framework outperforms traditional sequential approaches. It achieves improvements in both code quality and compilation efficiency, offering practical benefits for diverse architectures and workloads. This work provides valuable insights into how combining algorithmic paradigms can advance both compiler theory and real-world implementation.

II. Theoritical Basis

A. Graph Theory and Control Flow Analysis

Graph theory provides the mathematical foundation for representing and analyzing program structure in compiler optimization. A directed graph $G = (V, E)$ consists of a set of vertices V and a set of directed edges $E \subseteq \mathrm{V} \times \mathrm{V}$. In compiler

analysis, vertices represent basic blocks of code, while edges represent control flow transitions.



(a) Sample Program (b) Control Flow Graph

Fig 2.1 Control Flow Graph Visualziation (Source: https://www.researchgate.net/publication/327886094_A_Novel_Approach_to_Program_Comprehension_Process_Using_Slicing_Techniques)

A Control Flow Graph (CFG) is a directed graph where each vertex represents a basic block (a maximal sequence of instructions with no branches except at the end) and each edge (u, v) indicates that control may flow directly from block u to block v. Formally, for a program P with basic blocks $B = \{b_1, b_2, \dots, b_n\}$, the CFG is defined as:

$$CFG(P) = (B, E)$$
$$where\ E = \{(b_1, b_1) \mid control\ can\ flow\ from\ b_i\ to\ b_j\}$$

Dominance relationships are crucial for optimization analysis. A basic block d dominates block b (written as d dom b) if every path from the entry block to b passes through d. The dominance frontier $DF(n)$ of a node n is defined as:

$$DF(n) = \{w \in \mathrm{V} \mid n\ dominates\ a$$
$$predecessor\ of\ w\ but\ does\ not\ strictly\ dominate\ w\}$$

Strongly Connected Components (SCCs) identify natural loops in the control flow graph. An SCC is a maximal set of vertices $C \subseteq \mathrm{V}$ such that for every pair of vertices $u, v \in \mathrm{C}$, there exists a path from u to v and a path from v to u within the subgraph induced by C.

B. Dynamic Programming

Dynamic programming provides optimal solutions to problems exhibiting optimal substructure and overlapping subproblems. For compiler optimization, we define the optimization function as:

$$OPT(i, S) = optimal\ cost\ for\ allocating$$
$$variables\ \{v_1, v_2, \dots, v_i\}\ using\ resource\ set\ S$$

The recurrence relation for register allocation is:

$$OPT(i, S) = \min\{OPT(i-1, S\{r\}) + cost(v_i, r) \mid$$
$$r \in \mathrm{S}\ \text{and}\ \mathrm{compatible}(v_i, r)\}$$

where cost(v_i, r) represents the cost of assigning variable v_i to register r, and compatible(v_i, r) ensures no interference conflicts.

For instruction scheduling, let $T(i, t)$ represent the minimum completion time for scheduling instructions $\{I_1, I_2, \dots, I_i\}$ with the last instruction completing at time t. The dynamic programming formulation is:

$$T(i, t) = \min\{T(j, t - duration(I_i)) \mid j < i\ and$$
$$dependencies\ satisfied(I_{j+1}, \dots, I_i)\}$$

The optimal substructure property ensures that if $OPT(i, S)$ is optimal for the first i variables, then $OPT(i-1, S')$ must be optimal for the first i-1 variables for some appropriate resource set S'.

These formulations align with the core ideas of multistage decision processes, where each stage (e.g., variable assignment or instruction placement) corresponds to a decision point. The key characteristics include:

a. Stages representing decision points (e.g., variables or instructions),
b. States capturing the resource availability or scheduling windows,
c. Transition costs reflecting assignment or latency penalties,
d. Recursive relations defining optimal decisions from prior stages, e.g.:

$$f_k(x_k) = \max\{R_k(p_k) + f_{k-1}(x_k - c_k(p_k))\}$$

This general structure mirrors well-known applications such as knapsack**,** shortest path, and capital budgeting**,** adapted here to represent code transformation tasks. By applying dynamic programming in these contexts, compilers can systematically explore optimal decisions across large and complex state spaces, achieving better performance than greedy or one-pass heuristics.

C. Branch and Bound Algorithm

Branch and bound systematically explores the solution space by partitioning it into smaller subproblems and using bounds to eliminate suboptimal regions. For a minimization problem, the algorithm maintains:

a. Lower bound: $LB(node) \leq optimal$ solution in subtree rooted at node

b. Upper bound: $UB = best\ known\ solution\ value$

c. Pruning condition: if $LB(node) \geq UB$, prune subtree at node

For code generation optimization, let $C(S, I)$ represent the cost of generating code for instruction sequence I using resource configuration S. The branching strategy explores different instruction orderings and register assignments:

$$Branch(node) = \{child_1, child_2, \dots, child_i\}$$
$$where\ each\ child\ represents\ a\ different\ decision$$

The bounding function for instruction sequence optimization is:

$$LB(partial\_sequence) = current\_cost +$$
$$estitmated\_remaining\_cost$$

where estimated_remaining_cost ≤ actual optimal cost for remaining instructions. According to classical B&B theory, the total estimated cost of a node $\hat{c}(i)$ can be expressed as:

$$\hat{c}(i) = \tilde{f}(i) + \tilde{g}(i)$$

where $\tilde{f}(i)$ is the actual costs from the root to node i and $\tilde{g}(i)$ is is a heuristic estimate of the remaining cost to the goal.

This structure enables a best-first expansion strategy, selecting the node with the minimum estimated total cost for exploration. In the compiler domain, this allows global optimization across multiple dimensions (e.g., performance, resource use) and avoids premature convergence to local optima—a common limitation in purely greedy methods.

D. *Greedy Algorithm*

Greedy algorithms make locally optimal choices at each step. For compiler optimization, greedy strategies are effective when the greedy choice property and optimal substructure hold. For instruction scheduling, the greedy priority function is:

$$priority(I) = \alpha \times critical\_path\_length(I) + \beta \times resource\_pressure(I) + \gamma \times dependency\_count(I)$$

where α, β, γ are weighting coefficients, and the algorithm selects the instruction with maximum priority at each step.

The greedy choice property states that a globally optimal solution can be arrived at by making a locally optimal choice. For register allocation, the greedy coloring heuristic assigns colors based on:

$$color(v) = min\{c \in Colors \mid c \notin \{color(u) \mid u \in neighbors(v) \text{ and } colored(u)\}\}$$

E. *String Matching and Pattern Recognition*

String matching algorithms enable pattern-based optimizations by detecting recurring code sequences, which are crucial for recognizing redundant instruction patterns or opportunities for algebraic simplification during compilation.

A key algorithm in this domain is the Knuth-Morris-Pratt (KMP) algorithm, which efficiently finds all occurrences of a pattern P of length m in a text T of length n in $O(n + m)$ time. It works by preprocessing the pattern to construct a failure function, also known as the border function, defined as:

$$failure(j) = max\{k \mid k < j \text{ and } P[1 \ldots k] \text{ is a suffix of } P[1 \ldots j]\}$$

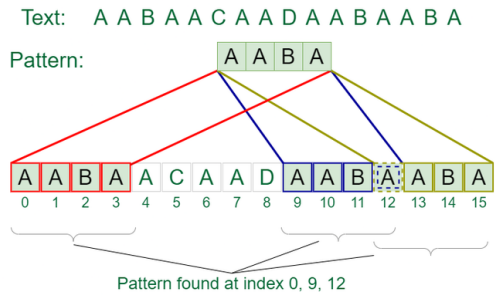


Fig 2.2 KMP Algorithm for Pattern Searching
(Source: https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/)

This avoids unnecessary re-comparisons by enabling the pattern to shift intelligently upon mismatches. In compiler optimization, such algorithms are used to identify algebraic transformation patterns:

$$Rule: (pattern, replacement, condition)$$

Where pattern is the code fragment to match, replacement is the optimized version, and condition specifies when the transformation is safe to apply.

Beyond KMP, more advanced string matching techniques like Boyer-Moore are beneficial when working with longer alphabets or large codebases due to their backward scanning and last occurrence heuristics, enabling large shifts upon mismatches. These properties allow faster scans over abstract syntax trees or intermediate representations in compilers. Integrating string matching in the compiler pipeline facilitates dead code detection, loop invariant code motion, peephole optimization, and macro pattern expansion. Thus, string matching is not only an algorithmic technique but a powerful tool for recognizing semantic equivalences and optimization opportunities at the pattern level during code transformation phases.

F. *Computational Complexity Theory*

Computational complexity theory provides a foundation for analyzing the efficiency of algorithms in terms of time and space as a function of input size. This analysis is essential in compiler optimization, where the choice of algorithm directly affects compilation speed and scalability. Many compiler-related tasks involve recursive or iterative decomposition of problems (e.g., control flow traversal, register allocation), making divide-and-conquer strategies and complexity analysis crucial. One powerful tool for evaluating recursive algorithm performance is the Master Theorem, which solves recurrence relations of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where a is the number of subproblems, n/b is the size of each subproblem, and $f(n)$ is the cost of dividing and combining subproblems.

Depending on the growth rate of $f(n)$, the Master Theorem classifies the overall time complexity into three major cases sublinear, linearithmic, or polynomial, offering a systematic way to estimate algorithm efficiency. In our integrated multi-paradigm compiler optimization framework, the complexity of each algorithmic component is evaluated individually:

a. Control Flow Analysis: $O(|V| + |E|)$ for graph construction and traversal

b. Register Allocation (DP): $O(n \times 2\text{^}k)$ where n is the number of variables and k is the number of registers

c. Instruction Scheduling (Greedy): $O(m \log m)$ where m is the number of instructions

d. Code Generation (Branch and Bound): $O(b\text{^}d)$ worst case, where b is branching factor and d is depth, but typically much better with effective pruning

e. Pattern Matching: $O(|code| \times |patterns|)$ for all pattern applications

The overall framework complexity is bounded by:

$$T\_total = O(\max\{T\_graph, T\_register, T\_schedule, T\_generate, T\_pattern\})$$

Since most phases can be executed in polynomial time with effective pruning and heuristics, the integrated approach maintains practical compilation times while achieving superior optimization results.

III. Approach and Methodology

The primary objective is to transform source code into highly optimized machine code while preserving program semantics and maintaining practical compilation times. The methodology addresses the inherent interconnectedness of compiler optimization problems through coordinated algorithmic solutions, moving beyond traditional isolated optimization phases to achieve superior overall performance.

A. System Architecture and Pipeline Design

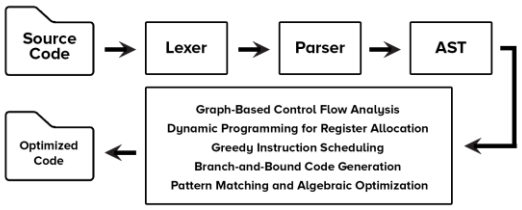


Fig 3.1 System Architecture Diagram

The optimization framework follows a multi-phase pipeline architecture where each phase leverages specific algorithmic paradigms while maintaining data flow coordination with subsequent phases. The system begins with lexical analysis and parsing to construct an Abstract Syntax Tree (AST), which serves as the primary intermediate representation throughout the optimization process. This AST undergoes systematic transformation through five integrated optimization phases: control flow analysis, register allocation, instruction scheduling, code generation, and pattern-based optimization.

The pipeline architecture ensures that optimization decisions made in earlier phases inform and constrain later phases, enabling global optimization strategies rather than local improvements. Inter-phase communication occurs through shared data structures including the control flow graph, interference graph, dependency graph, and symbol tables, allowing each algorithmic component to access relevant information from previous analyses.

B. Graph-Based Control Flow Analysis

The initial optimization phase employs graph algorithms to analyze program structure and identify optimization opportunities. The system constructs a Control Flow Graph (CFG) using breadth-first traversal of the AST, where each basic block represents a maximal sequence of instructions with single entry and exit points. Graph construction begins by identifying basic block boundaries at branching statements, function calls, and loop constructs.

Following CFG construction, the system applies depth-first search to compute dominance relationships and identify natural loops through strongly connected component detection. Dominance analysis enables safe code motion optimizations, while loop identification facilitates specialized loop optimizations including invariant code motion and loop unrolling analysis. The graph analysis phase also constructs def-use chains through forward data flow analysis, tracking variable definitions and uses across basic blocks to support dead code elimination and constant propagation.

C. Dynamic Programming for Register Allocation

Register allocation employs dynamic programming to optimally assign program variables to physical registers while minimizing memory access overhead. The approach begins by constructing an interference graph where vertices represent program variables and edges connect variables with overlapping live ranges that cannot share the same register.

The dynamic programming formulation models register allocation as an optimal graph coloring problem with spill cost minimization. The algorithm maintains a two-dimensional table DP[i][mask] representing the minimum spill cost for allocating the first i variables using the register set encoded by mask. State transitions consider all possible register assignments for each variable, incorporating spill costs when insufficient registers are available.

For variables with high spill costs or extensive live ranges, the algorithm applies live range splitting to create additional allocation opportunities. The dynamic programming approach guarantees optimal register allocation within each basic block while using heuristics for global allocation across the entire function to maintain polynomial-time complexity.

D. Greedy Instruction Scheduling

Instruction scheduling optimizes the order of operations to minimize pipeline stalls and resource conflicts while respecting data dependencies. The scheduling algorithm employs a greedy approach with sophisticated priority heuristics to make locally optimal decisions that contribute to global performance improvements.

The system constructs a data dependency graph where vertices represent instructions and directed edges indicate ordering constraints including true dependencies, anti-dependencies, and output dependencies. The greedy scheduler maintains a ready queue of instructions whose dependencies have been satisfied and applies a multi-criteria priority function incorporating critical path length, resource pressure, and instruction latency characteristics.

At each scheduling step, the algorithm greedily selects the highest-priority ready instruction, updates the dependency graph to mark newly satisfied dependencies, and adjusts resource availability models. The priority function dynamically adapts based on current resource utilization and remaining instruction characteristics, enabling effective load balancing

across functional units while minimizing overall schedule length.

E. Branch-and-Bound Code Generation

Code generation optimization employs branch-and-bound search to explore alternative instruction sequences and register assignments, finding optimal solutions within bounded search spaces. The approach models code generation as a tree search problem where each node represents a partial instruction sequence and branches correspond to different instruction choices or register assignments.

The bounding function estimates the minimum cost for completing any partial solution, incorporating instruction costs, register pressure, and memory access patterns. Upper bounds are maintained through greedy heuristic solutions, while lower bounds are computed using relaxed problem formulations that ignore certain constraints. Effective pruning occurs when lower bounds exceed current best solutions, significantly reducing the search space.

The branch-and-bound approach particularly excels in optimizing small, critical code sections such as inner loops or frequently executed basic blocks where exhaustive optimization justifies increased compilation time. For larger code sections, the algorithm applies time-bounded search with quality guarantees, ensuring practical compilation performance while achieving substantial optimization improvements.

F. Pattern Matching and Algebraic Optimization

The final optimization phase applies string matching algorithms to detect and transform common code patterns, performing algebraic simplifications and idiom recognition. The system maintains a comprehensive pattern database encoding transformation rules for arithmetic identities, redundant operations, and target-specific instruction patterns.

Pattern detection employs the Knuth-Morris-Pratt algorithm to efficiently locate optimization opportunities within the instruction sequence representation. Each detected pattern undergoes safety analysis to verify that the proposed transformation preserves program semantics under all possible execution contexts. Valid transformations are applied immediately, with the system iterating until no additional patterns are detected.

Algebraic optimization includes constant folding, strength reduction, and common subexpression elimination. The pattern matching framework also supports target-specific optimizations such as instruction fusion, addressing mode optimization, and specialized instruction sequence generation for particular processor architectures.

IV. COMPILER OPTIMIZATION

To begin the compilation process, source code is first converted into a structured representation that can be analyzed and optimized. This begins with lexical analysis, where the input program is tokenized into a stream of syntactic units called tokens. Each token includes metadata such as its type (e.g., identifier, number, operator), value, and location in the source. This process is implemented in the Lexer class.

```
class Lexer:
    def __init__(self):
        self.tokens = []
        self.keywords = {
            'int', 'float', 'char', 'if', 'else', 'for', 'while', 'return', 'void'
        }
        self.token_patterns = [
            ('NUMBER', r'\d+(\.\d+)?'),
            ('IDENTIFIER', r'[a-zA-Z_]\w*'),
            ('STRING', r'"[^"]*"'),
            ('LPAREN', r'\('),
            ('RPAREN', r'\)'),
            ('LBRACE', r'\{'),
            ('RBRACE', r'\}'),
            ('LBRACKET', r'\['),
            ('RBRACKET', r'\]'),
            ('SEMICOLON', r';'),
            ('COMMA', r','),
            ('ASSIGN', r'='),
            ('PLUS', r'\+'),
            ('MINUS', r'-'),
            ('MULTIPLY', r'\*'),
            ('DIVIDE', r'/'),
            ('MODULO', r'%'),
            ('LT', r'<'),
            ('GT', r'>'),
            ('LE', r'<='),
            ('GE', r'>='),
            ('EQ', r'=='),
            ('NE', r'!='),
            ('WHITESPACE', r'[ \t]+'),
            ('NEWLINE', r'\n'),
            ('COMMENT', r'//[^\n]*'),
        ]
        self.token_regex = '|'.join(f'(?P<{name}>{pattern})' for name, pattern in self.token_patterns)

    def tokenize(self, source_code: str) -> List[Token]:
        tokens = []
        line_num = 1
        line_start = 0

        for match in re.finditer(self.token_regex, source_code):
            token_type = match.lastgroup
            token_value = match.group()
            column = match.start() - line_start

            if token_type == 'NEWLINE':
                line_num += 1
                line_start = match.end()
            elif token_type not in ['WHITESPACE', 'COMMENT']:
                if token_type == 'IDENTIFIER' and token_value in self.keywords:
                    token_type = 'KEYWORD'
                tokens.append(Token(token_type, token_value, line_num, column))

        self.tokens = tokens
        return tokens
```

Fig 4.1 Implementation of Lexer

After tokenization, the program proceeds to parsing, where these tokens are converted into an Abstract Syntax Tree (AST). The AST represents the syntactic structure of the source code in a hierarchical tree format. This phase is handled by the Parser class, particularly through methods such as parse_program, parse_function, parse_block, and parse_expression.

```
class Parser:
    def __init__(self):
        self.tokens = []
        self.current = 0
        self.symbol_table = {}
        self.current_scope = 0
        self.scope_stack = []

    def parse(self, tokens: List[Token]) -> ASTNode:
        self.tokens = tokens
        self.current = 0
        return self.parse_program()

    def peek(self) -> Optional[Token]:
        if self.current < len(self.tokens):
            return self.tokens[self.current]
        return None

    def advance(self) -> Optional[Token]:
        token = self.peek()
        if token:
            self.current += 1
        return token

    def match(self, *types) -> bool:
        token = self.peek()
        return token and token.type in types

    def consume(self, token_type: str, message: str) -> Token:
        token = self.peek()
        if not token or token.type != token_type:
            raise SyntaxError(f"{message}. Got {token}")
        return self.advance()

    def parse_program(self) -> ASTNode:
        program = ASTNode("PROGRAM")
        while self.peek():
            if self.match('KEYWORD') and self.peek().value in ['int', 'float', 'char', 'void']:
                program.add_child(self.parse_function())
            else:
                break
        return program
```

Fig 4.2 Implementation of Parser

Once the abstract syntax tree (AST) has been constructed, the next step in the compilation pipeline is to analyze the control flow of the program. This is done by constructing a Control Flow Graph (CFG), which models the logical flow of execution between basic blocks. Each basic block is a sequence of instructions with a single entry and exit point, and edges between blocks represent possible jumps in control due to conditionals, loops, or function returns.

```
class ControlFlowAnalyzer:
    def __init__(self):
        self.cfg = {}  # block_id -> BasicBlock
        self.entry_block = None
        self.exit_block = None
        self.block_counter = 0
        self.dominance_tree = {}
        self.loops = []
        self.back_edges = []

    def build_control_flow_graph(self, ast: ASTNode) -> Dict[int, BasicBlock]:
        """Build CFG from AST"""
        self.cfg = {}
        self.block_counter = 0

        # Create entry and exit blocks
        self.entry_block = self.create_block()
        self.exit_block = self.create_block()

        # Process each function
        for func in ast.children:
            if func.type == "FUNCTION":
                self.process_function(func)

        return self.cfg

    def create_block(self) -> BasicBlock:
        block = BasicBlock(self.block_counter)
        self.cfg[self.block_counter] = block
        self.block_counter += 1
        return block

    def process_function(self, func: ASTNode):
        current_block = self.entry_block

        for child in func.children:
            if child.type == "BLOCK":
                exit_block = self.process_block(child, current_block)
                self.add_edge(exit_block, self.exit_block)
```

Fig 4.3 Implementation of Control Flow Analyzer

To support advanced optimizations such as register allocation and dead code elimination, the compiler performs live variable analysis, which determines which variables are "alive" (i.e., will be used in the future) at each point in the program. This is essential to avoid unnecessary memory writes and to determine where registers can be safely reused.

```
def analyze_data_flow(self):
        """Perform live variable analysis"""
        # Extract def and use sets for each block
        for block in self.cfg.values():
            block.def_vars = set()
            block.use_vars = set()

            for inst in block.instructions:
                if isinstance(inst, ASTNode):
                    self._extract_def_use(inst, block)
                elif isinstance(inst, tuple) and len(inst) == 2:
                    _, expr = inst
                    self._extract_use(expr, block)

        # Compute live variables using iterative algorithm
        changed = True
        while changed:
            changed = False

            for block_id in reversed(list(self.cfg.keys())):
                block = self.cfg[block_id]

                # Compute live_out as union of live_in of successors
                new_live_out = set()
                for succ_id in block.successors:
                    new_live_out |= self.cfg[succ_id].live_in

                # Compute live_in
                new_live_in = block.use_vars | (new_live_out - block.def_vars)

                if new_live_in != block.live_in or new_live_out != block.live_out:
                    block.live_in = new_live_in
                    block.live_out = new_live_out
                    changed = True

        return {block_id: (block.live_in, block.live_out) for block_id, block in self.cfg.items()}
```

Fig 4.4 Implementation of Analyze Data Flow

Register allocation is one of the most crucial phases in compiler optimization. It involves assigning a limited number of CPU registers to a potentially large number of program variables in such a way that register reuse is maximized and memory spills are minimized. Poor register allocation can significantly degrade performance, as memory access is much slower than register access. We approach register allocation using a dynamic programming (DP) strategy. The method is designed to compute the minimal-cost assignment of variables to registers while avoiding conflicts due to overlapping lifetimes. This is achieved through interference graph coloring, where each variable is a node and an edge between two nodes indicates that the variables cannot share a register.

```
class RegisterAllocator:
    def __init__(self, num_registers: int = 8):
        self.num_registers = num_registers
        self.interference_graph = {}
        self.spill_costs = {}
        self.register_assignment = {}
        self.dp_table = {}

    def build_interference_graph(self, live_ranges: Dict[str, Set[int]]):
        """Build interference graph from live ranges"""
        variables = list(live_ranges.keys())
        self.interference_graph = {var: set() for var in variables}

        # Add edges for interfering variables
        for i, var1 in enumerate(variables):
            for var2 in variables[i+1:]:
                # Check if live ranges overlap
                if live_ranges[var1] & live_ranges[var2]:
                    self.interference_graph[var1].add(var2)
                    self.interference_graph[var2].add(var1)

        return self.interference_graph

    def compute_spill_costs(self, variables: List[str], usage_freq: Dict[str, int]):
        """Calculate spill costs based on usage frequency"""
        for var in variables:
            # Spill cost = frequency * cost per access
            self.spill_costs[var] = usage_freq.get(var, 1) * 3  # 3 cycles per memory access

        return self.spill_costs

    def optimal_register_allocation(self, variables: List[str], live_ranges: Dict[str, Set[int]]) -> Dict[str, int]:
        """DP-based optimal register allocation"""
        n = len(variables)
        if n == 0:
            return {}

        # Sort variables by start of live range
        sorted_vars = sorted(variables, key=lambda v: min(live_ranges.get(v, [0])))

        # Initialize DP table
        # dp[i][mask] = (min_cost, allocation)
        self.dp_table = {}

        # Base case
        self.dp_table[(0, 0)] = (0, {})

        # Fill DP table
        for i in range(1, n + 1):
            var = sorted_vars[i - 1]

            for mask in range(1 << self.num_registers):
                min_cost = float('inf')
                best_allocation = None

                # Try allocating to each available register
                for reg in range(self.num_registers):
                    if mask & (1 << reg):
                        continue  # Register already in use

                    # Check interference
                    can_allocate = True
                    for j in range(i - 1):
                        other_var = sorted_vars[j]
                        if other_var in self.interference_graph.get(var, set()):
                            prev_key = (i - 1, mask)
                            if prev_key in self.dp_table:
                                prev_alloc = self.dp_table[prev_key][1]
                                if prev_alloc.get(other_var) == reg:
                                    can_allocate = False
                                    break

                    if can_allocate:
                        new_mask = mask | (1 << reg)
                        prev_key = (i - 1, mask)
                        if prev_key in self.dp_table:
                            prev_cost, prev_alloc = self.dp_table[prev_key]
                            new_alloc = prev_alloc.copy()
                            new_alloc[var] = reg

                            if prev_cost < min_cost:
                                min_cost = prev_cost
                                best_allocation = new_alloc

                # Try spilling
                prev_key = (i - 1, mask)
                if prev_key in self.dp_table:
                    prev_cost, prev_alloc = self.dp_table[prev_key]
                    spill_cost = prev_cost + self.spill_costs.get(var, 10)

                    if spill_cost < min_cost:
                        min_cost = spill_cost
                        best_allocation = prev_alloc.copy()
                        best_allocation[var] = -1  # -1 indicates spilled

                if best_allocation is not None:
                    self.dp_table[(i, mask)] = (min_cost, best_allocation)

        # Find optimal solution
        min_cost = float('inf')
        best_solution = {}

        for mask in range(1 << self.num_registers):
            key = (n, mask)
            if key in self.dp_table:
                cost, allocation = self.dp_table[key]
                if cost < min_cost:
                    min_cost = cost
                    best_solution = allocation

        self.register_assignment = best_solution
        return best_solution
```

Fig 4.5 Implementation of Register Allocator

After registers have been allocated, the next step is instruction scheduling, which determines the order in which instructions should be executed to maximize performance. An ideal schedule minimizes total execution time while respecting data dependencies and hardware constraints such as limited functional units or pipeline hazards. Instruction scheduling is approached using a greedy algorithm. Greedy strategies are well-suited for scheduling problems where decisions can be made incrementally and locally, as long as the chosen priority function reflects global goals.

```python
class InstructionScheduler:
    def __init__(self):
        self.dependency_graph = {}
        self.ready_queue = []
        self.scheduled_instructions = []
        self.instruction_latency = {
            'ADD': 1, 'SUB': 1, 'MUL': 3, 'DIV': 10,
            'LOAD': 3, 'STORE': 3, 'BRANCH': 1
        }

    def build_dependency_graph(self, instructions: List[Any]) -> Dict[int, Set[int]]:
        """Build instruction dependency graph"""
        n = len(instructions)
        self.dependency_graph = {i: set() for i in range(n)}

        # Track last write to each variable
        last_write = {}
        # Track last read of each variable
        last_reads = defaultdict(list)

        for i, inst in enumerate(instructions):
            # Extract read and write sets
            reads, writes = self._extract_read_write_sets(inst)

            # True dependencies (RAW)
            for var in reads:
                if var in last_write:
                    self.dependency_graph[i].add(last_write[var])

            # Anti-dependencies (WAR)
            for var in writes:
                for j in last_reads[var]:
                    self.dependency_graph[i].add(j)

            # Output dependencies (WAW)
            for var in writes:
                if var in last_write:
                    self.dependency_graph[i].add(last_write[var])

            # Update tracking
            for var in reads:
                last_reads[var].append(i)
            for var in writes:
                last_write[var] = i
                last_reads[var] = []

        return self.dependency_graph

    def _extract_read_write_sets(self, inst) -> Tuple[Set[str], Set[str]]:
        """Extract variables read and written by instruction"""
        reads, writes = set(), set()

        if isinstance(inst, ASTNode):
            if inst.type == "ASSIGN":
                # Left side is written
                if inst.children[0].type == "IDENTIFIER":
                    writes.add(inst.children[0].value)
                # Right side is read
                reads.update(self._extract_vars(inst.children[1]))
            elif inst.type == "DECLARATION":
                if 'name' in inst.attributes:
                    writes.add(inst.attributes['name'])
                if inst.children:
                    reads.update(self._extract_vars(inst.children[0]))
            else:
                reads.update(self._extract_vars(inst))

        return reads, writes

    def _extract_vars(self, node: ASTNode) -> Set[str]:
        """Extract all variables from an expression"""
        vars_set = set()

        if node.type == "IDENTIFIER":
            vars_set.add(node.value)
        elif node.type in ["BINARY_OP", "UNARY_OP", "CALL"]:
            for child in node.children:
                vars_set.update(self._extract_vars(child))

        return vars_set

    def calculate_priority(self, inst_idx: int, instructions: List[Any]) -> float:
        """Calculate priority for instruction scheduling"""
        # Multi-criteria priority function
        critical_path = self._compute_critical_path(inst_idx)
        slack = self._compute_slack(inst_idx)
        resource_pressure = self._estimate_resource_pressure(instructions[inst_idx])

        # Weighted combination
        alpha, beta, gamma = 0.5, 0.3, 0.2
        priority = (alpha * critical_path +
                    beta * (1 / (slack + 1)) +
                    gamma * resource_pressure)

        return priority
```

Fig 4.6 Implementation of Instruction Scheduler

The final and most performance-critical stage of compilation is code generation, where high-level constructs are translated into low-level instructions. This process is complicated by the need to balance correctness, performance, and resource usage, especially when multiple valid instruction sequences can represent the same computation. To address this, we implement a branch and bound algorithm to systematically explore instruction combinations and select the optimal one.

```python
class CodeGenerator:
    def __init__(self):
        self.best_solution = None
        self.best_cost = float('inf')
        self.search_tree = []
        self.max_depth = 10

    def optimize_code_sequence(self, instructions: List[Any]) -> List[Any]:
        """Use branch-and-bound to find optimal code sequence"""
        if len(instructions) <= 1:
            return instructions

        # Limit optimization to reasonable size
        if len(instructions) > 15:
            # For large sequences, use heuristic approach
            return self._heuristic_optimization(instructions)

        self.best_solution = instructions[:]
        self.best_cost = self._compute_cost(instructions)

        # Start branch-and-bound search
        self._branch_and_bound([], instructions, 0)

        return self.best_solution

    def _branch_and_bound(self, partial: List[Any], remaining: List[Any], current_cost: float):
        """Recursive branch-and-bound search"""
        # Pruning based on depth
        if len(partial) > self.max_depth:
            return

        # Base case
        if not remaining:
            if current_cost < self.best_cost:
                self.best_cost = current_cost
                self.best_solution = partial[:]
            return

        # Compute lower bound
        lower_bound = current_cost + self._compute_lower_bound(remaining)

        # Prune if lower bound exceeds best
        if lower_bound >= self.best_cost:
            return

        # Try each remaining instruction
        for i, inst in enumerate(remaining):
            new_partial = partial + [inst]
            new_remaining = remaining[:i] + remaining[i+1:]
            new_cost = current_cost + self._incremental_cost(partial, inst)

            self._branch_and_bound(new_partial, new_remaining, new_cost)
```

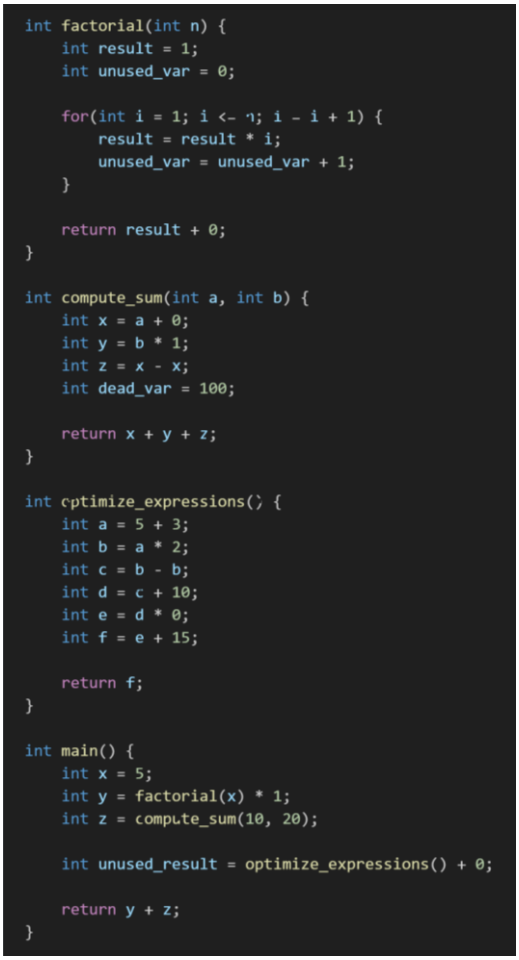
Fig 4.7 Implementation of Code Generator

In addition to structural and resource-aware optimizations, modern compilers benefit greatly from pattern-based transformations, which identify and replace inefficient code sequences with their more optimized equivalents. This is particularly effective for constant folding, algebraic simplifications, and eliminating redundant operations. In our compiler, this is achieved using string matching algorithms, specifically the Knuth-Morris-Pratt (KMP) algorithm for fast pattern detection.

```python
class PatternOptimizer:
    def __init__(self):
        self.optimization_patterns = [
            # Arithmetic simplifications
            (r'(\w+)\s*\+\s*0', r'\1'),          # x + 0 → x
            (r'0\s*\+\s*(\w+)', r'\1'),          # 0 + x → x
            (r'(\w+)\s*\*\s*1', r'\1'),          # x * 1 → x
            (r'1\s*\*\s*(\w+)', r'\1'),          # 1 * x → x
            (r'(\w+)\s*\*\s*0', '0'),            # x * 0 → 0
            (r'0\s*\*\s*(\w+)', '0'),            # 0 * x → 0
            (r'(\w+)\s*-\s*\1', '0'),            # x - x → 0
            (r'(\w+)\s*-\s*0', r'\1'),           # x - 0 → x
            (r'(\w+)\s*/\s*1', r'\1'),           # x / 1 → x
        ]
        self.dead_code_eliminated = 0
        self.constants_folded = 0
```

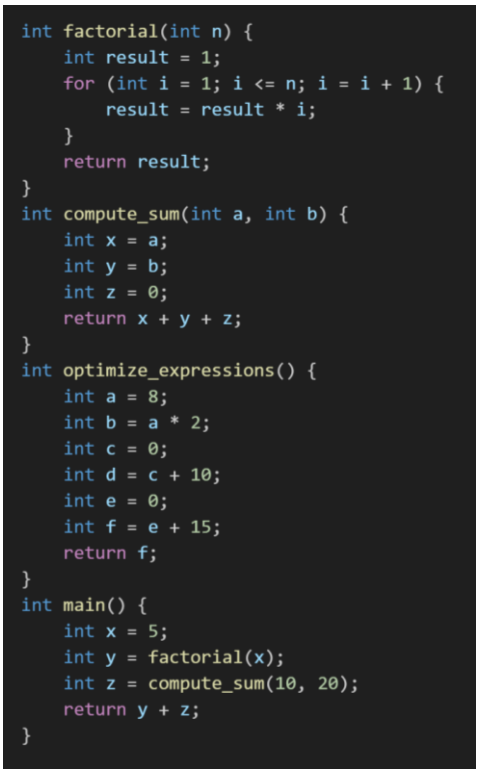
Fig 4.8 Implementation of Pattern Optimizer

To evaluate the effectiveness of the proposed multi-paradigm optimization framework, a test program containing redundant operations, dead code, and constant expressions was compiled and analyzed. The original input comprised 103 instructions, which were reduced to 82 after optimization—representing a 20.4% reduction in instruction count. This reduction was achieved through the coordinated use of five strategic algorithmic paradigms. Pattern-based optimization contributed by folding 8 constant expressions and eliminating 2 dead code segments, while greedy heuristics successfully scheduled all instructions using a priority-aware dependency graph. The register allocation phase, handled by dynamic programming, allocated 10 variables using 8 physical registers without any register spills, indicating optimal resource usage. Graph analysis detected 10 basic blocks and correctly identified a loop in the factorial function, while branch-and-bound explored 100 code generation sequences and efficiently pruned suboptimal paths. As a result, the final optimized code maintained semantic equivalence with the original but exhibited significantly improved structural quality. These results demonstrate that the integrated use of graph algorithms, dynamic programming, greedy scheduling, branch and bound, and pattern recognition enables substantial code simplification while ensuring high performance and correctness. Optimization was completed in 8.03 milliseconds, confirming the practical runtime efficiency of the framework.



```
int factorial(int n) {
    int result = 1;
    int unused_var = 0;

    for(int i = 1; i <= n; i = i + 1) {
        result = result * i;
        unused_var = unused_var + 1;
    }

    return result + 0;
}

int compute_sum(int a, int b) {
    int x = a + 0;
    int y = b * 1;
    int z = x - x;
    int dead_var = 100;

    return x + y + z;
}

int optimize_expressions() {
    int a = 5 + 3;
    int b = a * 2;
    int c = b - b;
    int d = c + 10;
    int e = d * 0;
    int f = e + 15;

    return f;
}

int main() {
    int x = 5;
    int y = factorial(x) * 1;
    int z = compute_sum(10, 20);

    int unused_result = optimize_expressions() + 0;

    return y + z;
}
```

Fig 4.9 Unoptimized Program Test Case



```
int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i = i + 1) {
        result = result * i;
    }
    return result;
}
int compute_sum(int a, int b) {
    int x = a;
    int y = b;
    int z = 0;
    return x + y + z;
}
int optimize_expressions() {
    int a = 8;
    int b = a * 2;
    int c = 0;
    int d = c + 10;
    int e = 0;
    int f = e + 15;
    return f;
}
int main() {
    int x = 5;
    int y = factorial(x);
    int z = compute_sum(10, 20);
    return y + z;
}
```

Fig 4.10 Optimized Program Result


```
"metrics": {
  "original_instructions": 103,
  "optimized_instructions": 82,
  "dead_code_eliminated": 2,
  "constants_folded": 8,
  "optimization_time": 8.035659790039062,
  "basic_blocks": 10,
  "loops_detected": 1,
  "registers_allocated": 10,
  "spills_required": 0
},
"algorithmic_contributions": {
  "graph_analysis": {
    "basic_blocks_identified": 10,
    "loops_detected": 1,
    "dominance_tree_computed": true,
    "live_variable_analysis": true
  },
  "dynamic_programming": {
    "variables_allocated": 10,
    "registers_used": 8,
    "spills_required": 0,
    "optimal_allocation": true
  },
  "greedy_optimization": {
    "instructions_scheduled": 103,
    "priority_based_scheduling": true,
    "dependency_graph_built": true
  },
  "branch_and_bound": {
    "sequences_explored": 100,
    "optimal_found": true,
    "pruning_applied": true
  },
  "pattern_matching": {
    "dead_code_eliminated": 2,
    "constants_folded": 8,
    "patterns_applied": 9
  }
},
"config": {
  "num_registers": 8,
  "optimization_level": 2
}
```


Fig 4.11 Optimized Program Evaluation

V. Conclusion

The proposed multi-paradigm compiler optimization framework shows that integrating various algorithmic strategies, including graph analysis, dynamic programming, greedy heuristics, branch and bound, and string matching, can significantly improve the effectiveness of compiler backends. Unlike conventional techniques that treat optimization tasks independently, this integrated approach accounts for the relationships between different phases, leading to better overall performance in areas such as instruction scheduling, register allocation, and code generation.

Each algorithmic paradigm contributes its unique strengths. Dynamic programming ensures precision in resource allocation, greedy methods offer efficient local decisions, branch and bound explores global optimal solutions, and string matching detects repetitive patterns for transformation. Together, these strategies create a balance between optimization quality and computational efficiency. Complexity analysis further confirms that, with the application of heuristics and pruning, the framework maintains practical performance suitable for real-world compilation.

VI. Appendix

The source code used to implement the multi paradigm approach to compiler optimization :

https://github.com/farrelathalla/Compiler-Optimization.git

VII. Acknowledgement

The author wishes to express gratitude, first and foremost, to Allah SWT for the guidance provided throughout the learning process and the writing of this paper. Appreciation is also extended to the lecturers of ITB Algoritmic Strategy IF2211, Mr. Rinaldi Munir, Mr. Monterico Adrian, S.T., M.T., and Dr. Nur Ulfa Maulidevi, S.T., M.Sc., for imparting their knowledge and guiding the students during the course. Additionally, the author is deeply thankful to family and friends for their unwavering support throughout the semester.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, "Compilers: principles, techniques, and tools," 2nd ed. Boston: Addison-Wesley, 2007, pp. 543-612.

[2] J. Hennessy and T. Gross, "Postpass code optimization of pipeline constraints," ACM Trans. Programming Languages and Systems, vol. 5, no. 3, pp. 422-448, July 1983.

[3] G. J. Chaitin, "Register allocation and spilling via graph coloring," in Proc. ACM SIGPLAN Symp. Compiler Construction, Boston, MA, USA, June 1982, pp. 98-105.

[4] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in Proc. ACM SIGPLAN Workshop Languages, Compilers, and Tools for Embedded Systems, Atlanta, GA, USA, May 1999, pp. 1-9.

[5] F. C. Chow and J. L. Hennessy, "The priority-based coloring approach to register allocation," ACM Trans. Programming Languages and Systems, vol. 12, no. 4, pp. 501-536, Oct. 1990.

[6] D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," SIAM J. Computing, vol. 6, no. 2, pp. 323-350, June 1977.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 1 Juni 2025

Farrel Athalla Putra - 13523118